US009189398B2

(12) **United States Patent** (10) **Patent No.:** **US 9,189,398 B2**
Pardo et al. (45) **Date of Patent:** **Nov. 17, 2015**

(54) **APPARATUS AND METHOD FOR MEMORY-MAPPED REGISTER CACHING**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Ilan Pardo**, Ramat-Hasharon (IL); **Michael Behar**, Zichron Yaakov (IL); **Oren Ben-Kiki**, Tel-Aviv (IL); **Dror Markovich**, Tel Aviv (IL)

(73) Assignee: **INTEL CORPORATION**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 217 days.

(21) Appl. No.: **13/730,030**

(22) Filed: **Dec. 28, 2012**

(65) **Prior Publication Data**

US 2014/0189191 A1 Jul. 3, 2014

(51) **Int. Cl.**
G06F 12/00 (2006.01)
G06F 12/08 (2006.01)
G06F 13/00 (2006.01)

(52) **U.S. Cl.**
CPC *G06F 12/0802* (2013.01); *G06F 12/0875* (2013.01); *G06F 12/0897* (2013.01); *Y02B 60/1225* (2013.01)

(58) **Field of Classification Search**
CPC G06F 12/00; G06F 12/10; G06F 12/0897
USPC 711/100, 117, 118, 122, 200, 202
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,918,245 A *	6/1999	Yung	....	711/122
6,711,653 B1 *	3/2004	Quach et al.	....	711/146
8,244,979 B2 *	8/2012	Burns et al.	....	711/122
2002/0144081 A1 *	10/2002	Willis et al.	....	711/206
2005/0091459 A1 *	4/2005	Quach et al.	....	711/141
2007/0005933 A1 *	1/2007	Kopec et al.	....	711/207
2013/0080734 A1 *	3/2013	Sato et al.	....	711/207

OTHER PUBLICATIONS

Gebhart, et al., "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors", 38th International Symposium on Computer Architecture, Jun. 4-8, 2011, 12 pages.
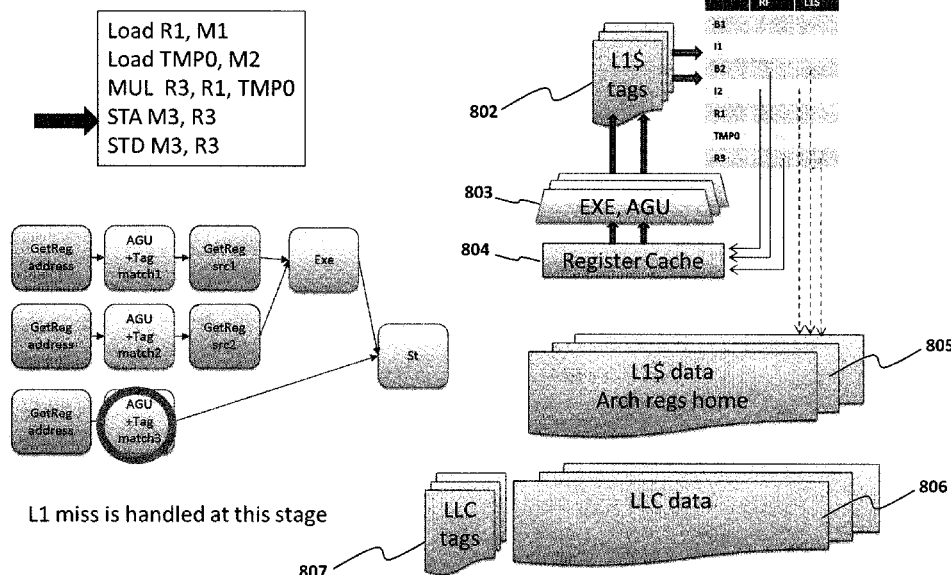
* cited by examiner

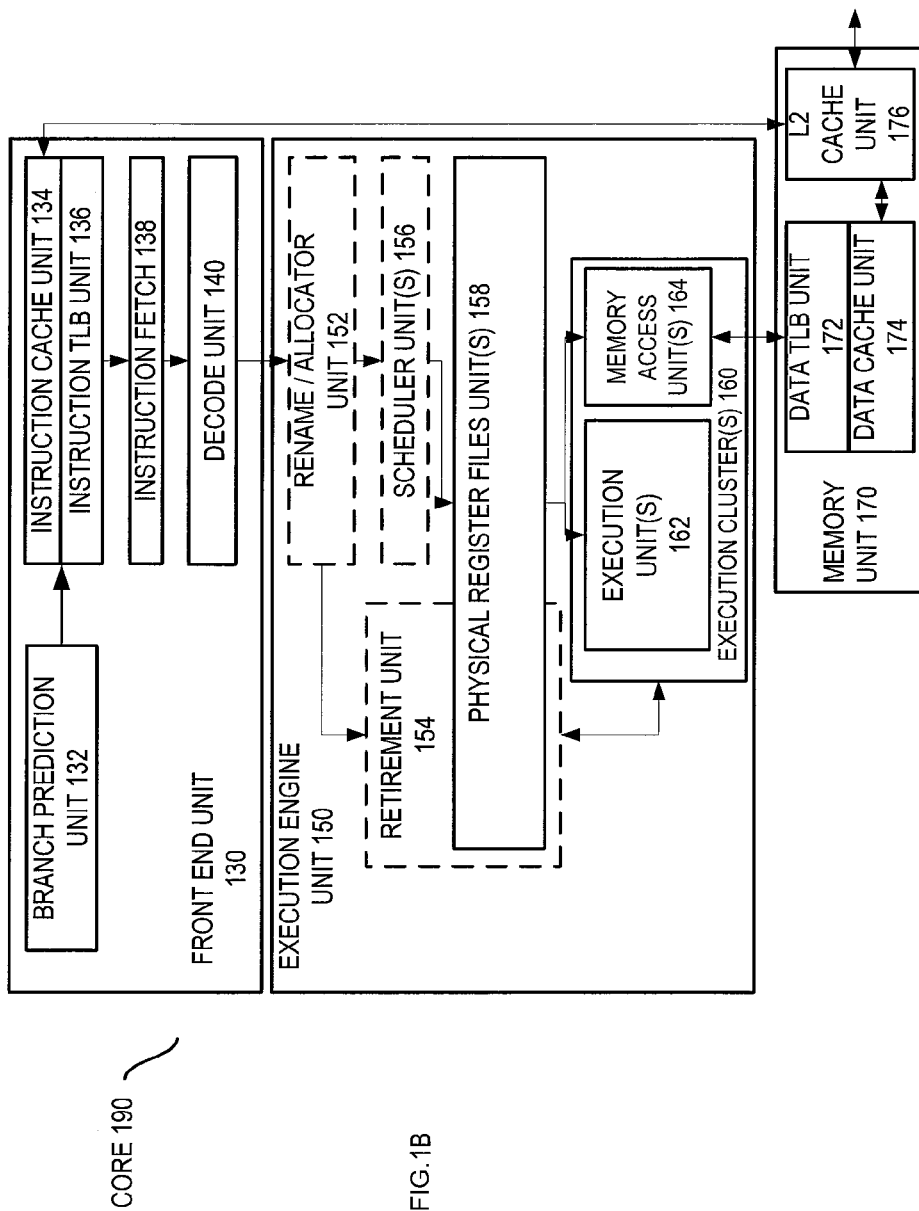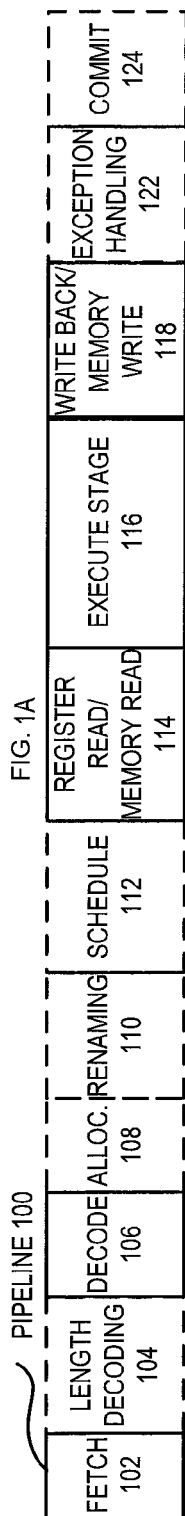*Primary Examiner* — Tuan Thai

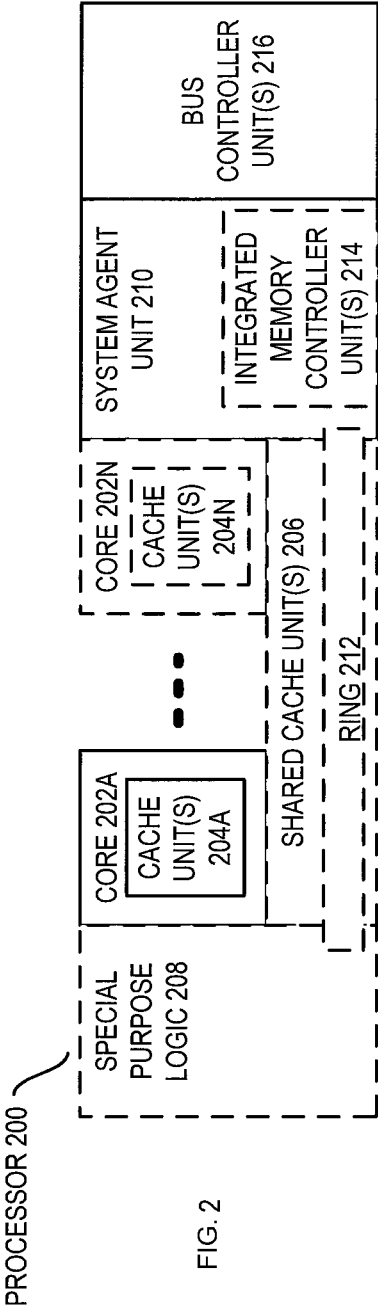(74) *Attorney, Agent, or Firm* — Nicholson De Vos Webster & Elliott LLP

(57) **ABSTRACT**

A processor is described comprising: an architectural register file implemented as a combination of a register file cache and an architectural register region within a level 1 (L1) data cache, and a data location table (DLT) to store data indicating a location of each architectural register within the register file cache and/or the architectural register region within the L1 data cache.
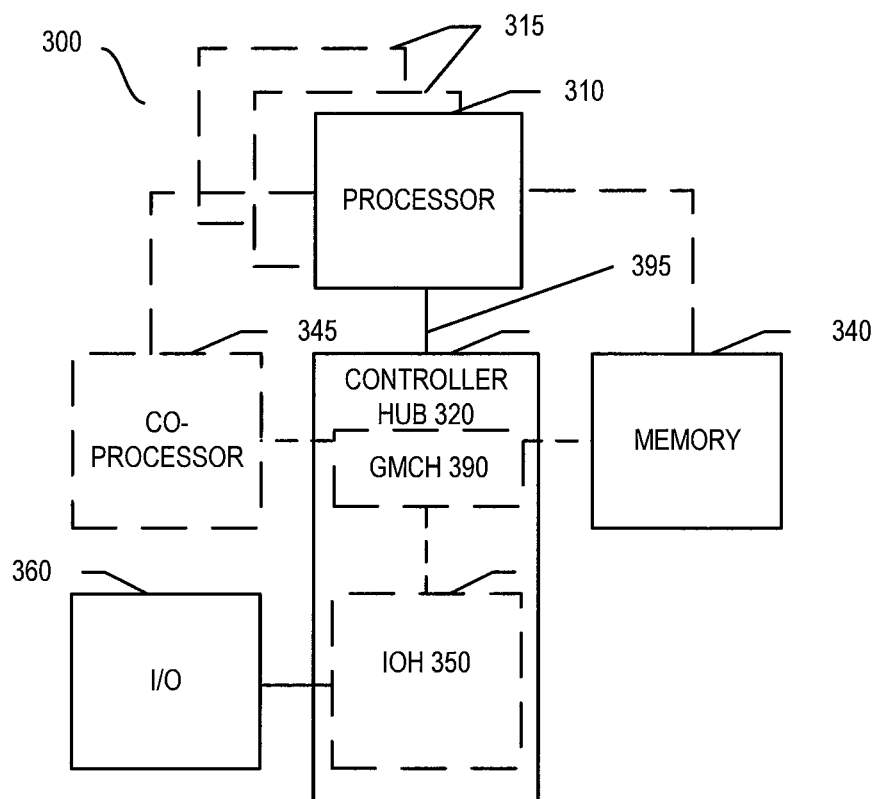
13 Claims, 21 Drawing Sheets

FIG. 1A

PIPELINE 100

| FETCH 102 | LENGTH DECODING 104 | DECODE 106 | ALLOC. 108 | RENAMING 110 | SCHEDULE 112 | REGISTER READ/ MEMORY READ 114 | EXECUTE STAGE 116 | WRITE BACK/ MEMORY WRITE 118 | EXCEPTION HANDLING 122 | COMMIT 124 |

FIG. 1B

CORE 190

FRONT END UNIT 130

BRANCH PREDICTION UNIT 132

INSTRUCTION CACHE UNIT 134

INSTRUCTION TLB UNIT 136

INSTRUCTION FETCH 138

DECODE UNIT 140

EXECUTION ENGINE UNIT 150

RENAME / ALLOCATOR UNIT 152

RETIREMENT UNIT 154

SCHEDULER UNIT(S) 156

PHYSICAL REGISTER FILES UNIT(S) 158

EXECUTION CLUSTER(S) 160

EXECUTION UNIT(S) 162

MEMORY ACCESS UNIT(S) 164

MEMORY UNIT 170

DATA TLB UNIT 172

DATA CACHE UNIT 174

L2 CACHE UNIT 176

FIG. 2

FIG. 3

FIG. 4

FIG. 5

FIG. 6

FIG. 7

PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 716

X86 BINARY CODE 706

X86 COMPILER 704

HIGH LEVEL LANGUAGE 702

HARDWARE

SOFTWARE

INSTRUCTION CONVERTER 712

PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 714

ALTERNATIVE INSTRUCTION SET BINARY CODE 710

ALTERNATIVE INSTRUCTION SET COMPILER 708

Fig. 8

Load R1, Base1, index1
Load TMP0, Base2, index2
MUL R3, R1, TMP0
STA M3, R3
STD M3, R3

Fig. 9

Fig. 10

Fig. 11

Load R1, M1
Load TMP0, M2
MUL R3, R1, TMP0
STA M3, R3
STD M3, R3

Dst allocation to Register Cache in DLT
is done at this stage

Fig. 12

Fig. 13

Load R1, M1
Load TMP0, M2
MUL R3, R1, TMP0
STA M3, R3
STD M3, R3

L1 miss is handled at this stage

Fig. 14

	RF	L1$
B1		
I1		
B2		
I2		
R1		
TMP0		
R3		

L1$ tags — 802

EXE, AGU — 803

Register Cache — 804

L1$ data — 805
Arch regs home

LLC data — 806

LLC tags — 807

Load R1, M1
Load TMP0, M2
MUL R3, R1, TMP0
STA M3, R3
STD M3, R3

GetReg
address → AGU +Tag match1 → GetReg src1 → Exe

GetReg
address → AGU +Tag match2 → GetReg src2

GetReg
address → AGU +Tag match3

St

Fig. 15

FIG. 16A

FIG. 16B

FULL OPCODE FIELD 1174

AUGMENTATION OPERATION FIELD 1150

| FORMAT FIELD 1140 | BASE OPERATION FIELD 1142 | REGISTER INDEX FIELD 1144 | MODIFIER FIELD 1146 | CLASS FIELD 1168 | ALPHA FIELD 1152 | BETA FIELD 1154 | SCALE FIELD 1160 | DISP. F. 1162A / DISP. F. F. 1162B | DATA ELEMENT WIDTH FIELD 1164 | WRITE MASK FIELD 1170 | IMMEDIATE FIELD 1172 |

RL FIELD 1157A

NO MEMORY ACCESS 1105

NO MEM. ACC., W.M.C., PART. RND. CNTRL. TYPE OP. 1112

| FORMAT FIELD 1140 | BASE OPERATION FIELD 1142 | REGISTER INDEX FIELD 1144 | NO MEMORY ACCESS 1146A | CLASS B 1168B | WRITE MASK CONTROL FIELD 1152C | RND 1157A.1 | ROUND OPERATION FIELD 1159A | | | DATA ELEMENT WIDTH FIELD 1164 | WRITE MASK FIELD 1170 | IMMEDIATE FIELD 1172 |

NO MEM. ACC., W.M.C., VSIZE TYPE OP. 1117

| FORMAT FIELD 1140 | BASE OPERATION FIELD 1142 | REGISTER INDEX FIELD 1144 | NO MEMORY ACCESS 1146A | CLASS B 1168B | WRITE MASK CONTROL FIELD 1152C | VSIZE 1157A.2 | VECTOR LENGTH FIELD 1159B | | | DATA ELEMENT WIDTH FIELD 1164 | WRITE MASK FIELD 1170 | IMMEDIATE FIELD 1172 |

MEMORY ACCESS 1120

MEM. ACC., W.M.C., 1127

| FORMAT FIELD 1140 | BASE OPERATION FIELD 1142 | REGISTER INDEX FIELD 1144 | MEMORY ACCESS 1146B | CLASS B 1168B | WRITE MASK CONTROL FIELD 1152C | BROAD CAST FIELD 1157B | VECTOR LENGTH FIELD 1159B | SCALE FIELD 1160 | DISP. F. 1162A / DISP. F. F. 1162B | DATA ELEMENT WIDTH FIELD 1164 | WRITE MASK FIELD 1170 | IMMEDIATE FIELD 1172 |

FIG. 17A

FIG. 17B

FIG. 17C

FIG. 17D

CLASS FIELD 1168   ALPHA FIELD 1152   BETA FIELD 1154

AUGMENTATION OPERATION FIELD 1150 | U | α | β | β | β |

U=0

RS FIELD 1152A | α | β | β | β | MOD FIELD 1242 | 11 | RS FIELD 1152A | α | β | β | β |

ROUND 1152A.1

| 1 | r_2 | r_1 | r_0 |

SAE FIELD 1156

ROUND OPERATION FIELD 1158

ROUND CONTROL FIELD 1154A

DATA TRANSFORM 1152A.2 | 0 | s_2 | s_1 | s_0 |

DATA TRANSFORM FIELD 1154B

| α | β | β | β | MOD FIELD 1242 | 00 | OR | 01 | OR | 10 |

1162A

EVICTION HINT FIELD 1152B | EH | s_2 | s_1 | s_0 | | SIB | | D | D | D | D |

1250   1162B

DATA MANIPULATION FIELD 1154C

U=1

WRITE MASK CONTROL FIELD 1152C | α |

MERGING | 0 |

ZEROING | 1 |

MOD FIELD 1242

| β | β | β | RL FIELD 1157A | 11 | | β | β | β | RL FIELD 1157A

| r_1 | r_0 | 1 | ROUND 1157A.1

ROUND OPERATION FIELD 1159A

| L_1 | L_0 | 0 | VSIZE 1157A.2

VECTOR LENGTH FIELD 1159B

MOD FIELD 1242

| β | β | β | | 00 | OR | 01 | OR | 10 |

1162A

| L_1 | L_0 | B | | SIB | | D | D | D | D |

1250   1162B

VECTOR LENGTH FIELD 1159B   BROADCAST FIELD 1157B

FIG. 18

SCALAR FP STACK REGISTER FILE
1345
(X87FP)
80 BITS

0

ALIASED

7

64 BITS

MMX PACKED INT FLAT
REGISTER FILE 1350

Write Mask Registers 1315
64 BITS

k_0

k_7

REGISTER ARCHITECTURE 1300

General Purpose Registers 1325
16 X 64 BITS

Vector Registers 1310
512 BITS

xmm_0

ymm_0

zmm_0

xmm_{15}

ymm_{15}

zmm_{31}

128 BITS

256 BITS

1900

Monitor 1905

Video Ouput Interface 1904

GPU(s) 1903

Graphics Interface 1902

Storage Device 1919

Storage Interface 1918

Memory Modules 1925

Memory Interface 1920

Accelerator Cluster(s) 801

Processor Cluster(s) 804

Interconnect(s)/bus(es) 1901

Power Management Interface 1917

Wireless Interface(s) (e.g., 802.11, Bluetooth) 1910

Cellular Interface 1911

Ethernet Interface 1912

Serial Bus Interface 1913

Camera Interface 1914

Sensor Hub 1915

Sensors 1909

Audio Input Interface 1916

FIG. 19

APPARATUS AND METHOD FOR MEMORY-MAPPED REGISTER CACHING

BACKGROUND

1. Field of the Invention

This invention relates generally to the field of computer processors. More particularly, the invention relates to an apparatus and method for memory-mapped register caching.

2. Description of the Related Art

Current throughput processing implementations use highly threaded processor architectures to compensate for memory access and execution latencies. With this approach, each thread has its own set of registers within a dedicated Main Register File (MRF), yielding an enormous register file which consumes a significant amount of power.

In current implementations, the MRF is not directly related to the L1 data cache. Consequently, whenever there is a load operation, data must first be read from the memory subsystem (possibly from the cache) and placed into the MRF, consuming processing cycles and space within the MRF.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

FIG. 1A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention;

FIG. 1B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention;

FIG. 2 is a block diagram of a single core processor and a multicore processor with integrated memory controller and graphics according to embodiments of the invention;

FIG. 3 illustrates a block diagram of a system in accordance with one embodiment of the present invention;

FIG. 4 illustrates a block diagram of a second system in accordance with an embodiment of the present invention;

FIG. 5 illustrates a block diagram of a third system in accordance with an embodiment of the present invention;

FIG. 6 illustrates a block diagram of a system on a chip (SoC) in accordance with an embodiment of the present invention;

FIG. 7 illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention;

FIGS. 8-15 illustrate an exemplary architecture for implementing memory-mapped register caching.

FIGS. 16A and 16B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

FIGS. 17A-D are a block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention; and

FIG. 18 is a block diagram of a register architecture according to one embodiment of the invention.

FIG. 19 illustrates a computer system in accordance with certain embodiments of the invention.

DETAILED DESCRIPTION

In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide

a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

Exemplary Processor Architectures and Data Types

FIG. 1A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. FIG. 1B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in FIGS. 1A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In FIG. 1A, a processor pipeline 100 includes a fetch stage 102, a length decode stage 104, a decode stage 106, an allocation stage 108, a renaming stage 110, a scheduling (also known as a dispatch or issue) stage 112, a register read/memory read stage 114, an execute stage 116, a write back/memory write stage 118, an exception handling stage 122, and a commit stage 124.

FIG. 1B shows processor core 190 including a front end unit 130 coupled to an execution engine unit 150, and both are coupled to a memory unit 170. The core 190 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 190 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit 130 includes a branch prediction unit 132 coupled to an instruction cache unit 134, which is coupled to an instruction translation lookaside buffer (TLB) 136, which is coupled to an instruction fetch unit 138, which is coupled to a decode unit 140. The decode unit 140 (or decoder) may decode instructions, and generate as an output one or more micro-operations, microcode entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 140 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 190 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 140 or otherwise within the front end unit 130). The decode unit 140 is coupled to a rename/allocator unit 152 in the execution engine unit 150.

The execution engine unit 150 includes the rename/allocator unit 152 coupled to a retirement unit 154 and a set of one or more scheduler unit(s) 156. The scheduler unit(s) 156 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 156 is coupled to the physical register file(s) unit(s) 158. Each of the physical register file(s) units 158 represents one or more physical register files, different ones of which store one or more different data types, such as scalar

integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit **158** comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) **158** is overlapped by the retirement unit **154** to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit **154** and the physical register file(s) unit(s) **158** are coupled to the execution cluster(s) **160**. The execution cluster(s) **160** includes a set of one or more execution units **162** and a set of one or more memory access units **164**. The execution units **162** may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. The scheduler unit(s) **156**, physical register file(s) unit(s) **158**, and execution cluster(s) **160** are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster—and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) **164**). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units **164** is coupled to the memory unit **170**, which includes a data TLB unit **172** coupled to a data cache unit **174** coupled to a level 2 (L2) cache unit **176**. In one exemplary embodiment, the memory access units **164** may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit **172** in the memory unit **170**. The instruction cache unit **134** is further coupled to a level 2 (L2) cache unit **176** in the memory unit **170**. The L2 cache unit **176** is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline **100** as follows: 1) the instruction fetch **138** performs the fetch and length decoding stages **102** and **104**; 2) the decode unit **140** performs the decode stage **106**; 3) the rename/allocator unit **152** performs the allocation stage **108** and renaming stage **110**; 4) the scheduler unit(s) **156** performs the schedule stage **112**; 5) the physical register file(s) unit(s) **158** and the memory unit **170** perform the register read/memory read stage **114**; the execution cluster **160** perform the execute stage **116**; 6) the memory unit **170** and the physical register file(s) unit(s) **158** perform the write back/memory write stage **118**; 7) various units may be involved in the exception handling stage **122**; and 8) the retirement unit **154** and the physical register file(s) unit(s) **158** perform the commit stage **124**.

The core **190** may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, Calif.; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, Calif.), including the instruction(s) described herein. In one embodiment, the core **190** includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2, and/or some form of the generic vector friendly instruction format (U=0 and/or U=1), described below), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units **134/174** and a shared L2 cache unit **176**, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

FIG. **2** is a block diagram of a processor **200** that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in FIG. **2** illustrate a processor **200** with a single core **202**A, a system agent **210**, a set of one or more bus controller units **216**, while the optional addition of the dashed lined boxes illustrates an alternative processor **200** with multiple cores **202**A-N, a set of one or more integrated memory controller unit(s) **214** in the system agent unit **210**, and special purpose logic **208**.

Thus, different implementations of the processor **200** may include: 1) a CPU with the special purpose logic **208** being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores **202**A-N being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores **202**A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores **202**A-N being a large number of general purpose in-order cores. Thus, the processor **200** may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor **200** may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

5

The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units **206**, and external memory (not shown) coupled to the set of integrated memory controller units **214**. The set of shared cache units **206** may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit **212** interconnects the integrated graphics logic **208**, the set of shared cache units **206**, and the system agent unit **210**/integrated memory controller unit(s) **214**, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units **206** and cores **202-A-N**.

In some embodiments, one or more of the cores **202A-N** are capable of multi-threading. The system agent **210** includes those components coordinating and operating cores **202A-N**. The system agent unit **210** may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores **202A-N** and the integrated graphics logic **208**. The display unit is for driving one or more externally connected displays.

The cores **202A-N** may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores **202A-N** may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

FIGS. **3-6** are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to FIG. **3**, shown is a block diagram of a system **300** in accordance with one embodiment of the present invention. The system **300** may include one or more processors **310**, **315**, which are coupled to a controller hub **320**. In one embodiment the controller hub **320** includes a graphics memory controller hub (GMCH) **390** and an Input/Output Hub (IOH) **350** (which may be on separate chips); the GMCH **390** includes memory and graphics controllers to which are coupled memory **340** and a coprocessor **345**; the IOH **350** is couples input/output (I/O) devices **360** to the GMCH **390**. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory **340** and the coprocessor **345** are coupled directly to the processor **310**, and the controller hub **320** in a single chip with the IOH **350**.

The optional nature of additional processors **315** is denoted in FIG. **3** with broken lines. Each processor **310**, **315** may include one or more of the processing cores described herein and may be some version of the processor **200**.

The memory **340** may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub **320** communicates with the processor(s) **310**, **315** via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection **395**.

6

In one embodiment, the coprocessor **345** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub **320** may include an integrated graphics accelerator.

There can be a variety of differences between the physical resources **310**, **315** in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

In one embodiment, the processor **310** executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor **310** recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor **345**. Accordingly, the processor **310** issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor **345**. Coprocessor(s) **345** accept and execute the received coprocessor instructions.

Referring now to FIG. **4**, shown is a block diagram of a first more specific exemplary system **400** in accordance with an embodiment of the present invention. As shown in FIG. **4**, multiprocessor system **400** is a point-to-point interconnect system, and includes a first processor **470** and a second processor **480** coupled via a point-to-point interconnect **450**. Each of processors **470** and **480** may be some version of the processor **200**. In one embodiment of the invention, processors **470** and **480** are respectively processors **310** and **315**, while coprocessor **438** is coprocessor **345**. In another embodiment, processors **470** and **480** are respectively processor **310** coprocessor **345**.

Processors **470** and **480** are shown including integrated memory controller (IMC) units **472** and **482**, respectively. Processor **470** also includes as part of its bus controller units point-to-point (P-P) interfaces **476** and **478**; similarly, second processor **480** includes P-P interfaces **486** and **488**. Processors **470**, **480** may exchange information via a point-to-point (P-P) interface **450** using P-P interface circuits **478**, **488**. As shown in FIG. **4**, IMCs **472** and **482** couple the processors to respective memories, namely a memory **432** and a memory **434**, which may be portions of main memory locally attached to the respective processors.

Processors **470**, **480** may each exchange information with a chipset **490** via individual P-P interfaces **452**, **454** using point to point interface circuits **476**, **494**, **486**, **498**. Chipset **490** may optionally exchange information with the coprocessor **438** via a high-performance interface **439**. In one embodiment, the coprocessor **438** is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

Chipset **490** may be coupled to a first bus **416** via an interface **496**. In one embodiment, first bus **416** may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present invention is not so limited.

As shown in FIG. **4**, various I/O devices **414** may be coupled to first bus **416**, along with a bus bridge **418** which couples first bus **416** to a second bus **420**. In one embodiment, one or more additional processor(s) **415**, such as coproces-

sors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus **416**. In one embodiment, second bus **420** may be a low pin count (LPC) bus. Various devices may be coupled to a second bus **420** including, for example, a keyboard and/or mouse **422**, communication devices **427** and a storage unit **428** such as a disk drive or other mass storage device which may include instructions/code and data **430**, in one embodiment. Further, an audio I/O **424** may be coupled to the second bus **420**. Note that other architectures are possible. For example, instead of the point-to-point architecture of FIG. **4**, a system may implement a multi-drop bus or other such architecture.

Referring now to FIG. **5**, shown is a block diagram of a second more specific exemplary system **500** in accordance with an embodiment of the present invention. Like elements in FIGS. **4** and **5** bear like reference numerals, and certain aspects of FIG. **4** have been omitted from FIG. **5** in order to avoid obscuring other aspects of FIG. **5**.

FIG. **5** illustrates that the processors **470**, **480** may include integrated memory and I/O control logic ("CL") **472** and **482**, respectively. Thus, the CL **472**, **482** include integrated memory controller units and include I/O control logic. FIG. **5** illustrates that not only are the memories **432**, **434** coupled to the CL **472**, **482**, but also that I/O devices **514** are also coupled to the control logic **472**, **482**. Legacy I/O devices **515** are coupled to the chipset **490**.

Referring now to FIG. **6**, shown is a block diagram of a SoC **600** in accordance with an embodiment of the present invention. Similar elements in FIG. **2** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In FIG. **6**, an interconnect unit(s) **602** is coupled to: an application processor **610** which includes a set of one or more cores **202**A-N and shared cache unit(s) **206**; a system agent unit **210**; a bus controller unit(s) **216**; an integrated memory controller unit(s) **214**; a set or one or more coprocessors **620** which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit **630**; a direct memory access (DMA) unit **632**; and a display unit **640** for coupling to one or more external displays. In one embodiment, the coprocessor(s) **620** include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

Program code, such as code **430** illustrated in FIG. **4**, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if

desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as "IP cores" may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable's (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

FIG. **7** is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIG. **7** shows a program in a high level language **702** may be compiled using an x86 compiler **704** to generate x86 binary code **706** that may be natively executed by a processor with at least one x86 instruction set core **716**. The processor with at least one x86 instruction set core **716** represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler

704 represents a compiler that is operable to generate x86 binary code **706** (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core **716**. Similarly, FIG. 7 shows the program in the high level language **702** may be compiled using an alternative instruction set compiler **708** to generate alternative instruction set binary code **710** that may be natively executed by a processor without at least one x86 instruction set core **714** (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, Calif. and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, Calif.). The instruction converter **712** is used to convert the x86 binary code **706** into code that may be natively executed by the processor without an x86 instruction set core **714**. This converted code is not likely to be the same as the alternative instruction set binary code **710** because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter **712** represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code **706**.

Memory Mapped Register Caching

A register file must sustain high bandwidth data paths between itself and the functional units of the processor. To accomplish this goal, it must have a high port count, making the register file area large and power hungry. Because of the need to implement a large number of registers due to current highly threaded architectures, the register file is increased in size to a point which is either un-implementable or becomes a critical path that limits the processor's theoretical attainable frequency.

Prior work has been done to implement a relatively small register cache such that the small register cache supplies the needed bandwidth while the full register array bandwidth requirements are reduced. This scheme also reduces power since the cache access power is small relative to the power consumed for register array access. The number of cached registers is determined by the latency between a load and use of the loaded data by an execution command. Typically, several instructions separate the load operation from the execution command, thereby consuming space in the register file with data that is not used for several processor cycles.

The embodiments of the invention described below solve these and other problems. In one embodiment, the main register file (MRF) is implemented as a register file cache (RFC) and combined with the L1 data cache array. When a load command is executed, instead of moving data from the L1 data cache array, a tag lookup occurs and the physical location of the data in the data array is pointed to by a pointer in the data location table (DLT). Later, when the execution operation arrives, the data is moved directly into the RFC. This allows the RFC to be smaller since it does not need to hold the data for several processor cycles leading up to the actual use of the data by the execution operation. Moreover, the same data stored in the L1 data cache may be used by different threads. Modified register data is subsequently written back to the register area in the L1 data cache, in which case it is no longer associated with the memory cache line.

Additionally, the embodiments of the invention reduce the size of the register cache by fetching the data into the register cache just in time and reduce the number of data transfers from 3 transfers to 2 (as described in the example below). Register-to-register data movement is also eliminated by

changing the pointer in the DLT to where the data is currently stored (rather than actually moving the data). If a program or thread uses a subset of the registers at a certain time, then the memory addresses associated with the recently unused registers need not be contained in the processor storage. In this case, the cache data storage resource may be used for normal memory caching. The embodiments of the invention also allow for low overhead registers for saves and restores upon a context switch, using the same memory subsystem implementation serving the data cache and register shadow storage.

FIG. **8** illustrates the processor components used in one embodiment of the invention, including a plurality of execution units and address generation units **803**, a Data Location Table (DLT) **801**, a register cache **804** and a L1 data cache including a L1 data array **805** and L1 tag array **802**. Notably, no main register file (MRF) is used in this embodiment. As described below, the register cache **804** is used to provide the functions of the MRF but in a more efficient manner.

In one embodiment, the DLT **801** is implemented as a lookup table with an entry for every architectural register used by the instruction threads (maintained as a row in the table). For example, in an x86 implementation, the DLT contains entries for each x86 architectural register.

In one embodiment, the underlying data for each architectural register may be located in one of three locations, which may be identified by the DLT **801**:

    1) the L1 data architectural register area;

    2) the L1 data memory area; and/or

    3) the register cache **804**.

For each architectural register, the DLT holds pointers to the un-cached location of the register and, if cached, to the register cache location. In one embodiment, the register cache **804** does not rely on a tag structure as used in a standard data cache; it relies solely on the DLT to identify the register location.

One example of system operation will be described with respect to FIGS. **9** to **15** which show an exemplary flow of operations to execute a typical instruction sequence. In this particular example, two data elements located in memory are multiplied and the result is stored in a third memory location.

As illustrated in FIG. **9**, the DLT may be implemented as a lookup table with each row representing a particular architectural register (e.g., B1, B2, I1, I2, R1, TMP0, R3, etc). The first column (RF) provides a first pointer identifying the location of the architectural register data within the register cache **804** (if stored in the register cache) and a second column (L1$) provides a pointer to a location where the architectural register data is stored in the L1 data cache **805**. These pointers are continually updated so that the current location of the architectural register data can be readily identified. In one embodiment, the L1 pointers do not require a traditional lookup using tags. Rather, the L1 and register cache pointers are direct pointers to the set and way where the data is located (i.e., the physical address of the data). No additional layer of abstraction is used to identify the data, resulting in an extremely efficient lookup operation.

In the specific example shown in FIG. **9**, the initial operation requires calculating an address by combining the base and index values. In the example, M1 (memory location 1) is calculated using base1 and index1 and M2 is calculated using base2 and index2. In the example, the base and index values are not currently stored in the register cache **804**. As such, the DLT is referenced to identify the location of these values within the architectural register storage area of the L1 cache **805**. Thus, the pointer from the DLT is read and the values are moved from the L1 cache into the register cache **804** in order

to perform the memory calculations. The DLT may then be updated to reflect the new location of the values (within the register cache **804**).

As illustrated in FIG. **10**, an address generation unit **803** combines the base and index values now stored in the register cache **804** to generate an the two memory addresses M1 and M2. In the example shown in FIG. **10**, the L1 tags **802** are checked to determine whether the data is stored in the L1 cache. If there is a hit on the L1 tags **802**, this means that the data is already stored in the L1 cache and the physical address for the location of the data can be read from the L1 column in the DLT **801**. In such a case, this data is placed into the TMP0 and R1 architectural registers simply by changing the pointer to the data within the DLT **801** (i.e., the data is not physically moved—only the pointer for the TMP0 and R1 entries in the DLT are updated).

In the case of an L1 miss (i.e., the data is not in in the L1 cache **805**) the data must be located in a lower level cache (LLC) **806** (or from memory if not in the LLC). The arrows in FIG. **10** indicate that, in this example, the data is read from the LLC into the L1 cache. In such a case, the DLT is updated to reflect the new location of the data.

FIG. **11** illustrates the data for the multiply operation being loaded from the L1 cache **805** into the register cache **804** using the addresses M0 and M1 (determined as described above) just in time for the multiply operation (performed in FIG. **12**). Because the data is brought into the register cache **804** just in time, it can be made small compared to the main register files of current processors. For example, in current implementations, a compiler would cause the data to be stored in the register file or register cache several cycles prior to execution, which increases pressure on the cache. However, in the system described herein, the address phase is executed with the load operation, but the data is not brought into the register cache **804** until it is actually needed.

FIG. **12** illustrates the execution of the multiply operation using the data collected in the previous steps. For this operation, the source data in architectural registers R1 and TMP0 has been stored in the register cache **804**. The execution unit **803** performs the multiply operation using this data and stores the result in architectural register R3. The DLT **801** is updated with a new pointer to the register cache to reflect the new data stored in architectural register R3.

FIG. **13** begins the process to store the result of the multiply operation. The first step is to determine where the data will be stored (STA=store address and M3 is the memory address to which the data will be stored). The pointers for the base and index needed for the address calculation is transferred from the architectural area of the L1 cache into the register cache **804**. An address generation unit **803** combines the base and index values now stored in the register cache **804** to generate the memory address M3.

In the example shown in FIG. **14**, the L1 tags **802** are checked to determine whether the address is stored in the L1 cache. If there is a hit on the L1 tags **802**, this means that the data is already stored in the L1 cache and the physical address for the location of the data can be read from the L1 column in the DLT **801**. In such a case, this data is placed into the architectural registers of the register cache simply by changing the pointer to the data within the DLT **801** (i.e., the data is not physically moved—only the pointer in the DLT is updated). In the case of an L1 miss (i.e., the data is not in in the L1 cache **805**) the data must be located in a lower level cache (LLC) **806** (or from memory if not in the LLC).

FIG. **15** illustrates the store data operation, storing the data to the architectural register area of the L1 data cache **805**. The

pointer to the data is updated in the DLT **801** so that it may be readily identified in the L1 cache for subsequent operations.

The embodiments of the invention significantly reduce the size of the register file and are particularly useful when used in processors which execute single instruction multiple data (SIMD) instructions or other vector instruction types which pack multiple data elements into a single register. In traditional vector systems, the vector registers have a fixed size (e.g., 128 bits, 512 bits, etc) but, in some cases, the high order bits of the vector registers may not be used (depending on the element size and instruction). This results in an inefficient use of register space. The present invention solves this problem because a dedicated vector/SIMD register is not required for storing the data elements. Rather, the data elements are simply stored in the register cache and architectural register region of the L1 data cache and consume no more space than required. In other words, with a register cache such as described herein, only the data which is required is stored (and no space is unnecessarily wasted).

Another benefit of the above embodiments of the invention is that dedicated save/restore registers are not required on a context switch or other mechanism for saving and restoring thread state. Rather, the data is already mapped to memory so there is no need to explicitly save the context.

Exemplary Instruction Formats

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

FIGS. **16A-16B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. FIG. **16A** is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while FIG. **16B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format **1100** for which are defined class A and class B instruction templates, both of which include no memory access **1105** instruction templates and memory access **1120** instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4 byte) or 64 bit (8 byte) data element widths (or sizes) (and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may sup-

port more, less and/or different vector operand sizes (e.g., 256 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte) data element widths).

The class A instruction templates in FIG. **16**A include: 1) within the no memory access **1105** instruction templates there is shown a no memory access, full round control type operation **1110** instruction template and a no memory access, data transform type operation **1115** instruction template; and 2) within the memory access **1120** instruction templates there is shown a memory access, temporal **1125** instruction template and a memory access, non-temporal **1130** instruction template. The class B instruction templates in FIG. **16**B include: 1) within the no memory access **1105** instruction templates there is shown a no memory access, write mask control, partial round control type operation **1112** instruction template and a no memory access, write mask control, vsize type operation **1117** instruction template; and 2) within the memory access **1120** instruction templates there is shown a memory access, write mask control **1127** instruction template.

The generic vector friendly instruction format **1100** includes the following fields listed below in the order illustrated in FIGS. **17**A-17B.

Format field **1140**—a specific value (an instruction format identifier value) in this field uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in the vector friendly instruction format in instruction streams. As such, this field is optional in the sense that it is not needed for an instruction set that has only the generic vector friendly instruction format.

Base operation field **1142**—its content distinguishes different base operations.

Register index field **1144**—its content, directly or through address generation, specifies the locations of the source and destination operands, be they in registers or in memory. These include a sufficient number of bits to select N registers from a P×Q (e.g. 32×512, 16×128, 32×1024, 64×1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

Modifier field **1146**—its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is, between no memory access **1105** instruction templates and memory access **1120** instruction templates. Memory access operations read and/or write to the memory hierarchy (in some cases specifying the source and/or destination addresses using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

Augmentation operation field **1150**—its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field **1168**, an alpha field **1152**, and a beta field **1154**. The augmentation operation field **1150** allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

Scale field **1160**—its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses 2^{scale}*index+base).

Displacement Field **1162**A—its content is used as part of memory address generation (e.g., for address generation that uses 2^{scale}*index+base+displacement).

Displacement Factor Field **1162**B (note that the juxtaposition of displacement field **1162**A directly over displacement factor field **1162**B indicates one or the other is used)—its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N)—where N is the number of bytes in the memory access (e.g., for address generation that uses 2^{scale}*index+base+scaled displacement). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field **1174** (described herein) and the data manipulation field **1154**C. The displacement field **1162**A and the displacement factor field **1162**B are optional in the sense that they are not used for the no memory access **1105** instruction templates and/or different embodiments may implement only one or none of the two.

Data element width field **1164**—its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported using some aspect of the opcodes.

Write mask field **1170**—its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-writemasking, while class B instruction templates support both merging- and zeroing-writemasking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field **1170** allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the write mask field's **1170** content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's **1170** content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's **1170** content to directly specify the masking to be performed.

Immediate field **1172**—its content allows for the specification of an immediate. This field is optional in the sense that is it not present in an implementation of the generic vector

friendly format that does not support immediate and it is not present in instructions that do not use an immediate.

Class field **1168**—its content distinguishes between different classes of instructions. With reference to FIGS. **16A-B**, the contents of this field select between class A and class B instructions. In FIGS. **16A-B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A **1168**A and class B **1168**B for the class field **1168** respectively in FIGS. **16A-B**).

Instruction Templates of Class A

In the case of the non-memory access **1105** instruction templates of class A, the alpha field **1152** is interpreted as an RS field **1152**A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round **1152**A.**1** and data transform **1152**A.**2** are respectively specified for the no memory access, round type operation **1110** and the no memory access, data transform type operation **1115** instruction templates), while the beta field **1154** distinguishes which of the operations of the specified type is to be performed. In the no memory access **1105** instruction templates, the scale field **1160**, the displacement field **1162**A, and the displacement scale filed **1162**B are not present.

No-Memory Access Instruction Templates—Full Round Control Type Operation

In the no memory access full round control type operation **1110** instruction template, the beta field **1154** is interpreted as a round control field **1154**A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field **1154**A includes a suppress all floating point exceptions (SAE) field **1156** and a round operation control field **1158**, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field **1158**).

SAE field **1156**—its content distinguishes whether or not to disable the exception event reporting; when the SAE field's **1156** content indicates suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

Round operation control field **1158**—its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field **1158** allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's **1150** content overrides that register value.

No Memory Access Instruction Templates—Data Transform Type Operation

In the no memory access data transform type operation **1115** instruction template, the beta field **1154** is interpreted as a data transform field **1154**B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

In the case of a memory access **1120** instruction template of class A, the alpha field **1152** is interpreted as an eviction hint field **1152**B, whose content distinguishes which one of the eviction hints is to be used (in FIG. **16A**, temporal **1152**B.**1** and non-temporal **1152**B.**2** are respectively specified for the memory access, temporal **1125** instruction template and the memory access, non-temporal **1130** instruction template), while the beta field **1154** is interpreted as a data manipulation field **1154**C, whose content distinguishes which one of a number of data manipulation operations (also known as

primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access **1120** instruction templates include the scale field **1160**, and optionally the displacement field **1162**A or the displacement scale field **1162**B.

Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

Memory Access Instruction Templates—Temporal

Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates—Non-Temporal

Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

In the case of the instruction templates of class B, the alpha field **1152** is interpreted as a write mask control (Z) field **1152**C, whose content distinguishes whether the write masking controlled by the write mask field **1170** should be a merging or a zeroing.

In the case of the non-memory access **1105** instruction templates of class B, part of the beta field **1154** is interpreted as an RL field **1157**A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round **1157**A.**1** and vector length (VSIZE) **1157**A.**2** are respectively specified for the no memory access, write mask control, partial round control type operation **1112** instruction template and the no memory access, write mask control, VSIZE type operation **1117** instruction template), while the rest of the beta field **1154** distinguishes which of the operations of the specified type is to be performed. In the no memory access **1105** instruction templates, the scale field **1160**, the displacement field **1162**A, and the displacement scale filed **1162**B are not present.

In the no memory access, write mask control, partial round control type operation **1110** instruction template, the rest of the beta field **1154** is interpreted as a round operation field **1159**A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

Round operation control field **1159**A—just as round operation control field **1158**, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field **1159**A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's **1150** content overrides that register value.

In the no memory access, write mask control, VSIZE type operation **1117** instruction template, the rest of the beta field **1154** is interpreted as a vector length field **1159**B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

In the case of a memory access **1120** instruction template of class B, part of the beta field **1154** is interpreted as a broadcast field **1157**B, whose content distinguishes whether or not the

broadcast type data manipulation operation is to be performed, while the rest of the beta field 1154 is interpreted the vector length field 1159B. The memory access 1120 instruction templates include the scale field 1160, and optionally the displacement field 1162A or the displacement scale field 1162B.

With regard to the generic vector friendly instruction format 1100, a full opcode field 1174 is shown including the format field 1140, the base operation field 1142, and the data element width field 1164. While one embodiment is shown where the full opcode field 1174 includes all of these fields, the full opcode field 1174 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 1174 provides the operation code (opcode).

The augmentation operation field 1150, the data element width field 1164, and the write mask field 1170 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

The combination of write mask field and data element width field create typed instructions in that they allow the mask to be applied based on different data element widths.

The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

FIGS. 17A-B are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. FIG. 17A shows a specific vector friendly instruction format 1200 that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific vector friendly instruction format 1200 may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode

byte field, MOD R/M field, SIB field, displacement field, and immediate fields of the existing x86 instruction set with extensions. The fields from FIG. 17 into which the fields from FIG. 18 map are illustrated.

It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 1200 in the context of the generic vector friendly instruction format 1100 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 1200 except where claimed. For example, the generic vector friendly instruction format 1100 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 1200 is shown as having fields of specific sizes. By way of specific example, while the data element width field 1164 is illustrated as a one bit field in the specific vector friendly instruction format 1200, the invention is not so limited (that is, the generic vector friendly instruction format 1100 contemplates other sizes of the data element width field 1164).

The generic vector friendly instruction format 1100 includes the following fields listed below in the order illustrated in FIG. 17A.

EVEX Prefix (Bytes 0-3) 1202—is encoded in a four-byte form.

Format Field 1140 (EVEX Byte 0, bits [7:0])—the first byte (EVEX Byte 0) is the format field 1140 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

REX field 1205 (EVEX Byte 1, bits [7-5])—consists of a EVEX.R bit field (EVEX Byte 1, bit [7]-R), EVEX.X bit field (EVEX byte 1, bit [6]-X), and 1157BEX byte 1, bit[5]-B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

REX' field 1110—this is the first part of the REX' field 1110 and is the EVEX.R' bit field (EVEX Byte 1, bit [4]-R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the invention, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

Opcode map field 1215 (EVEX byte 1, bits [3:0]-mmmm)—its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

Data element width field 1164 (EVEX byte 2, bit [7]-W)—is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the datatype (either 32-bit data elements or 64-bit data elements).

EVEX.vvvv 1220 (EVEX Byte 2, bits [6:3]-vvvv)—the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions

with 2 or more source operands; 2) EVEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 1111 b. Thus, EVEX.vvvv field **1220** encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers.

EVEX.U **1168** Class field (EVEX byte 2, bit [2]-U)—If EVEX.0=0, it indicates class A or EVEX.U0; if EVEX.0=1, it indicates class B or EVEX.U1.

Prefix encoding field **1225** (EVEX byte 2, bits [1:0]-pp)—provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

Alpha field **1152** (EVEX byte 3, bit [7]-EH; also known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also illustrated with α)—as previously described, this field is context specific.

Beta field **1154** (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s$_{2-0}$, EVEX.r$_{2-0}$, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with βββ)—as previously described, this field is context specific.

REX' field **1110**—this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3]-V') that may be used to encode either the upper 16 or lower 16 of the extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V', EVEX.vvvv.

Write mask field **1170** (EVEX byte 3, bits [2:0]-kkk)—its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

Real Opcode Field **1230** (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field **1240** (Byte 5) includes MOD field **1242**, Reg field **1244**, and R/M field **1246**. As previously described, the MOD field's **1242** content distinguishes between memory access and non-memory access operations. The role of Reg field **1244** can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field **1246** may include the following: encoding the instruction operand

that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) Byte (Byte 6)—As previously described, the scale field's **1150** content is used for memory address generation. SIB.xxx **1254** and SIB.bbb **1256**—the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

Displacement field **1162**A (Bytes 7-10)—when MOD field **1242** contains 10, bytes 7-10 are the displacement field **1162**A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

Displacement factor field **1162**B (Byte 7)—when MOD field **1242** contains 01, byte 7 is the displacement factor field **1162**B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between −128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values −128, −64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field **1162**B is a reinterpretation of disp8; when using displacement factor field **1162**B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field **1162**B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field **1162**B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

Immediate field **1172** operates as previously described.

Full Opcode Field

FIG. **17**B is a block diagram illustrating the fields of the specific vector friendly instruction format **1200** that make up the full opcode field **1174** according to one embodiment of the invention. Specifically, the full opcode field **1174** includes the format field **1140**, the base operation field **1142**, and the data element width (W) field **1164**. The base operation field **1142** includes the prefix encoding field **1225**, the opcode map field **1215**, and the real opcode field **1230**.

Register Index Field

FIG. **17**C is a block diagram illustrating the fields of the specific vector friendly instruction format **1200** that make up the register index field **1144** according to one embodiment of the invention. Specifically, the register index field **1144** includes the REX field **1205**, the REX' field **1210**, the MODR/M.reg field **1244**, the MODR/M.r/m field **1246**, the VVVV field **1220**, xxx field **1254**, and the bbb field **1256**.

Augmentation Operation Field

FIG. **17**D is a block diagram illustrating the fields of the specific vector friendly instruction format **1200** that make up the augmentation operation field **1150** according to one embodiment of the invention. When the class (U) field **1168** contains 0, it signifies EVEX.U0 (class A **1168**A); when it contains 1, it signifies EVEX.U1 (class B **1168**B). When U=0

and the MOD field **1242** contains 11 (signifying a no memory access operation), the alpha field **1152** (EVEX byte 3, bit [7]-EH) is interpreted as the rs field **1152A**. When the rs field **1152A** contains a 1 (round **1152A.1**), the beta field **1154** (EVEX byte 3, bits [6:4]-SSS) is interpreted as the round control field **1154A**. The round control field **1154A** includes a one bit SAE field **1156** and a two bit round operation field **1158**. When the rs field **1152A** contains a 0 (data transform **1152A.2**), the beta field **1154** (EVEX byte 3, bits [6:4]-SSS) is interpreted as a three bit data transform field **1154B**. When U=0 and the MOD field **1242** contains 00, 01, or 10 (signifying a memory access operation), the alpha field **1152** (EVEX byte 3, bit [7]-EH) is interpreted as the eviction hint (EH) field **1152B** and the beta field **1154** (EVEX byte 3, bits [6:4]-SSS) is interpreted as a three bit data manipulation field **1154C**.

When U=1, the alpha field **1152** (EVEX byte 3, bit [7]-EH) is interpreted as the write mask control (Z) field **1152C**. When U=1 and the MOD field **1242** contains 11 (signifying a no memory access operation), part of the beta field **1154** (EVEX byte 3, bit [4]-S_0) is interpreted as the RL field **1157A**; when it contains a 1 (round **1157A.1**) the rest of the beta field **1154** (EVEX byte 3, bit [6-5]-S_{2-1}) is interpreted as the round operation field **1159A**, while when the RL field **1157A** contains a 0 (VSIZE **1157.A2**) the rest of the beta field **1154** (EVEX byte 3, bit [6-5]-S_{2-1}) is interpreted as the vector length field **1159B** (EVEX byte 3, bit [6-5]-L_{1-0}). When U=1 and the MOD field **1242** contains 00, 01, or 10 (signifying a memory access operation), the beta field **1154** (EVEX byte 3, bits [6:4]-SSS) is interpreted as the vector length field **1159B** (EVEX byte 3, bit [6-5]-L_{1-0}) and the broadcast field **1157B** (EVEX byte 3, bit [4]-B).

FIG. **18** is a block diagram of a register architecture **1300** according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers **1310** that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format **1200** operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 1159B	A (FIG. 11A; U = 0)	1110, 1115, 1125, 1130	zmm registers (the vector length is 64 byte)
	B (FIG. 11B; U = 1)	1112	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 1159B	B (FIG. 11B; U = 1)	1117, 1127	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 1159B

In other words, the vector length field **1159B** selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length; and instructions templates without the vector length field **1159B** operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format

1200 operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

Write mask registers **1315**—in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers **1315** are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

General-purpose registers **1325**—in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (x87 stack) **1345**, on which is aliased the MMX packed integer flat register file **1350**—in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

An Exemplary Computer System

FIG. **19** is a block diagram illustrating an exemplary clients and servers which may be used in some embodiments of the invention. It should be understood that while FIG. **19** illustrates various components of a computer system **1900**, it is not intended to represent any particular architecture or manner of interconnecting the components as such details are not germane to the embodiments of the invention. It will be appreciated that other computer systems that have fewer components or more components may also be used with the embodiments of invention.

As illustrated in FIG. **19**, the computer system **1900**, which is a form of a data processing system, includes the interconnect(s)/bus(es) **1901** communicatively coupling the processor cluster(s) **804** to the various other system components. The interconnects/buses may include various levels of interconnection which may be connected to each other through various bridges, controllers, and/or adapters as is well known in the art. By way of example, the interconnect(s) **1901** may include a quick path interconnect (QPI) component, a Peripheral Component Interconnect Express ("PCI Express") component, or other technologies for interconnecting the various components to the processor cluster(s) **804**. The underlying principles of the invention are not limited to any particular interconnects or buses.

Although illustrated as a separate component in FIG. **19**, the accelerator(s) **801** may be integrated within the processor cluster(s) **804**. Alternatively, some accelerator(s) may be integrated within the processor cluster(s) and some may be connected to the computer system via the interconnect(s)/bus(es). As described in detail above, the accelerators are adapted to efficiently execute certain types of program code

(e.g., vector/SIMD operations, graphics operations, sort and loop operations, etc). By way of example, the general purpose processor clusters **804** may include execution logic within a processor core for executing general purpose instructions such as x86 instructions including instructions which invoke commands on the accelerator clusters **801**. The underlying principles of the invention, however, are not limited to any particular type of general purpose clusters or accelerator clusters.

The embodiment illustrated in FIG. **19** also includes a memory interface **1920** for coupling memory modules **1925** computer system. In one embodiment, the memory modules **1925** are dual in-line memory modules (DIMMs) such as random access memory (RAM) modules and the memory interface may generate the electrical signaling required to access the memory modules **1925** (e.g., such as column address strobe (CAS), row address strobe (RAS), write enable (WE), and output enable (OE) signals).

In one embodiment, the memory interface **1920** comprises logic and circuitry for interfacing with different types of memory modules including volatile memory modules such as RAM and non-volatile memory modules such as Phase-Change Memory (PCM), also sometimes referred to as phase change random access memory (PRAM or PCRAM), PCME, Ovonic Unified Memory, or Chalcogenide RAM (C-RAM). For example, one embodiment of the computer system **1900** implements a two-level (2L) memory hierarchy comprising a "near memory" portion which may be a volatile memory such as RAM and a "far memory" portion which may be implemented as a Phase-Change Memory (PCM). In such a case, the memory interface may include the logic and circuitry required to access both memory types.

The illustrated embodiment **1900** also includes one or more storage interfaces **1918** for interfacing with storage devices such as hard drives or other non-volatile storage devices. In one embodiment, the storage interface **1918** comprises a serial ATA storage interface and the hard drive comprises a solid state drive (SSD) or a magnetic storage device. In an embodiment of the invention which uses 2LM memory (as discussed above), a portion of the storage on the storage device **1919** may be used for "far memory" (or a portion of "far memory").

The illustrated embodiment **1900** also includes a graphics interface **1902** for interfacing with one or more graphics processing units **1903**. The GPUs may be embedded on a motherboard of the computer system or on a separate card inserted in the motherboard (e.g., via a PCI express graphics interface, or other high speed graphics interface). A video output interface **1904** such as a digital video interface (DVI), High-Definition Multimedia Interface (HDMI), or Display-Port video output interface outputs a video stream to a monitor **1905** which renders video for the end user. As mentioned, the GPUs may be implemented as accelerator components for executing graphics program code using any of the embodiments described herein.

The illustrated embodiment **1900** also includes an audio input interface **1916** for receiving multiple digital and analog audio inputs. For example, a microphone may be coupled to one of the audio input interfaces to capture the user's voice (e.g., during Web chats, phone calls, or for recording audio). Additionally, a digital audio input may be used such as a Toslink interface.

The illustrated embodiment **1900** also includes a sensor hub **1915** for collecting data from various different system sensors **1909**. By way of example, and not limitation, the sensors **1909** may include mechanical sensors, motion sensors, and location sensors to detect a position and orientation

of the computer system **1900**. For example, in one embodiment, the sensors may include multi-axis accelerometers for detecting acceleration values along the X, Y, and Z axes and reporting to the data to the sensor hub. The sensor hub may then perform calculations to determine a current orientation of the computer system **1900**. For example, if the computer system is a notebook computer, the sensor hub may detect a current position of the computer monitor. The sensors **1909** may also include inertial sensors for detecting displacements from a reference location and/or proximity sensors for detecting proximity to a user or other device. In one embodiment, the sensors **1909** include a global positioning system (GPS) sensor or other sensor for determining the current global position of the computer system. The sensors **1909** may also include a magnetometer for detecting the orientation of the Earth's electric field (i.e., to determine a current position of the computing system relative to North). The sensors **1909** may also include a gyro for detecting changes in orientation and an ambient light sensor for detecting current lighting conditions (e.g., so that the sensor hub or other system component may responsively adjust the brightness of the monitor **1905**).

All of the data collected from the various sensors **1909** may be used to determine a current mode of operation and responsively adjust operation of the computing device **1900**. For example, in response to the signals from the sensors **1909** the computing device may enter into a first mode of operation in which in which the accelerator invocations described herein are enabled and a second mode of operation in which the accelerator invocations described herein are disabled.

The illustrated embodiment **1900** also includes a camera interface **1914** for coupling to a video camera usable to capture motion video and still pictures. For example, in one embodiment, the camera interface **1914** gathers motion video for video conferencing applications in which the accelerator invocation techniques described herein may be used. For example, one accelerator may be configured to efficiently encode video streams into the H.264/MPEG-4 AVC format. It should be noted, however, that the underlying principles of the invention are not limited to any particular video compression format.

The illustrated embodiment **1900** also includes a serial bus interface for establishing serial data communication with connected devices (e.g., mobile phones, tablets, printers, external cameras, MIDI devices, etc). This embodiment further includes an Ethernet interface **1912** for establishing network connections over an Ethernet network and a cellular interface **1911** for establishing voice and data connections over a cellular network using cellular communication protocols. Various cellular technologies may be employed including, but not limited to 3rd Generation Partnership Project technologies (e.g., 3GPP2) code division multiple access technologies (e.g., CDMA2000 technology using 1xRTT/EVDO/eHRPD); Long Term Evolution (LTE) technology and/or LTE-Advanced (LTE-A) technology; and Universal Mobile Telecommunications System (UMTS) technology such as WCDMA/TDSCDMA. In addition, the embodiment shown also includes a WiFi and/or Bluetooth interface **1910** for establishing communication over WiFi channels (e.g., 802.11 channels) and/or Bluetooth channels, respectively. Each of the Ethernet, Cellular, and WiFi communication interfaces include a transceiver and other circuitry for generating analog transmission signals using the appropriate technology. In one embodiment, an accelerator may also be invoked to support the network communication process (e.g., for performing network baseband functions such as data encoding).

The illustrated embodiment **1900** also includes a power management interface **1917** for detecting current conditions within the computer system (e.g., thermal, power usage, battery life, etc) and responsively adjusting power usage to each of the different system components. For example, under certain conditions, the the power management interface **1917** may turn off the accelerator functions described herein to conserve power (e.g., when the battery drops below a threshold value).

The illustrated embodiment **1900** also includes a power management interface **1917** may also include various different types of Input/Output devices such as a cursor control (e.g., mouse, touchscreen, touchpad, etc.), a keyboard, etc.) for receiving user input.

It will be appreciated that additional components, not shown in FIG. **19**, may also be a part of the data processing system **1900** in certain embodiments of the invention, and in certain embodiments of the invention fewer components than shown in FIG. **19** may be used. In addition, it will be appreciated that one or more buses and/or interconnects, not shown in FIG. **19**, may be used to interconnect the various components as is well known in the art.

Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals—such as carrier waves, infrared signals, digital signals, etc.). In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware. Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a

thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

What is claimed is:

1. A processor comprising:
an architectural register file implemented as a combination of a register file cache and an architectural register region within a level 1 (L1) data cache, wherein the architectural register file is further implemented to store data within a data memory area of the L1 data cache; and
a data location table (DLT) to store data indicating a location of each architectural register within the register file cache and/or the architectural register region within the L1 data cache,
wherein in response to detecting that data stored in the data memory area of the L1 data cache is needed for an instruction, data is transferred from the memory area of the L1 data cache to the register file cache and/or the architectural register region within the L1 data cache, and wherein the transfer to the register file cache is deferred until the data is actually needed for a computational instruction.

2. The processor as in claim **1** wherein the DLT comprises an entry for each architectural register of the architectural register file.

3. The processor as in claim **2** wherein the DLT comprises a first column containing pointers to the register file cache.

4. The processor as in claim **3** wherein the DLT comprises a second column containing pointers to the architectural register region within the L1 data cache.

5. The processor as in claim **1** further comprising:
a lower level cache (LLC) for storing data, wherein in response to detecting that the data needed for an instruction is not present in the register file cache and/or the architectural register region within the L1 data cache, data is transferred from the LLC to the register file cache and/or the architectural register region within the L1 data cache.

6. The processor as in claim **1** further comprising:
a tag cache to store L1 tag values, the tag cache to be queried to determine whether data needed for an instruction is stored within the L1 cache.

7. A method comprising:
implementing an architectural register file as a combination of a register file cache and an architectural register region within a level 1 (L1) data cache, and
storing data indicating a location of each architectural register within the register file cache and/or the architectural register region within the L1 data cache within a data location table (DLT) in response to detecting that data needed for an instruction is not present in the register file cache and/or the architectural register region within the L1 data cache, data being transferred from a lower level cache (LLC) to the register file cache and/or the architectural register region within the L1 data cache, wherein the transfer to the register file cache is deferred until the data is actually needed for a computational instruction.

8. The method as in claim **7** wherein the DLT comprises an entry for each architectural register of the architectural register file.

9. The method as in claim **8** wherein the DLT comprises a first column containing pointers to the register file cache.

10. The method as in claim **9** wherein the DLT comprises a second column containing pointers to the architectural register region within the L1 data cache.

11. The method as in claim **7** wherein the architectural register file is further implemented to store data within a data memory area of the L1 data cache.

12. The method as in claim **11** wherein, in response to detecting that data stored in the data memory area of the L1 data cache is needed for an instruction, data is transferred from the data memory area of the L1 data cache to the register file cache and/or the architectural register region within the L1 data cache.

13. The method as in claim **11** further comprising:

a tag cache to store L1 tag values, the tag cache to be queried to determine whether data needed for an instruction is stored within the L1 cache.

* * * * *